Introduction to SQL

Class 12

Course Overview

- Introduction to SQL
  - Databases, Tables
  - Classification of SQL – DDL, DML, DCL, TCL
    - DDL – CREATE, ALTER, DROP
    - DML – SELECT, INSERT, UPDATE, DELETE
    - DCL – GRANT, REVOKE
    - TCL – COMMIT, ROLLBACK, SAVEPOINT
  - Data types, Operators
  - Keys – Primary, Foreign, Composite, Unique, Alternate
  - Integrity Constraints – Domain Integrity Constrains, Entity Integrity Constraints, Referential Integrity Constraints
  - Joins – Outer Joins, Left Outer Joins, Right Outer Joins, Inner Joins.
  - Queries, Subqueries, Functions, Flow Control (IF, CASE, WHILE, REPEAT, LOOP), ,Stored functions ,Stored Procedures
  - Views
  - Indexes, Cursors, Triggers, Events
  - Concurrency and locking (Implicit locks, explicit locks, row level locks, table level locks, database level locks)
  - Tuning SQL queries and optimizing performance
  - SQL Databases vs NoSQL Databases
  - ACID, CAP
  - How SQL databases internally works

Views

SQL CREATE VIEW Example:

- SQL creates a view that shows all customers from Brazil:

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

- Query the view above as follows:

```
SELECT * FROM [Brazil Customers];
```

- SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

- Query the view above as follows:

```
SELECT * FROM [Products Above Average Price];
```

SQL Updating a View

- A view can be updated with the CREATE OR REPLACE VIEW statement.
  - ***CREATE OR REPLACE VIEW view_name AS***
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition;

  - *Example*- SQL adds the "City" column to the "Brazil Customers" view:
    - ***CREATE OR REPLACE VIEW [Brazil Customers] AS***
      SELECT CustomerName, ContactName, City
      FROM Customers
      WHERE Country = 'Brazil';

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

- The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
- The SELECT statement should not have the DISTINCT keyword.
- The View should have all NOT NULL values.
- The view should not be created using nested queries or complex queries.
- The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

- The **WITH CHECK OPTION** clause in SQL is a very useful clause for views. It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.
- The **WITH CHECK OPTION** clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
- If we have used the **WITH CHECK OPTION** clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.
- Example -

CREATE VIEW SampleView AS

SELECT S_ID, NAME

FROM StudentDetails

WHERE NAME IS NOT NULL

WITH CHECK OPTION;

SQL Dropping View

- A view is deleted with the DROP VIEW statement.

  - ***DROP VIEW view_name;***

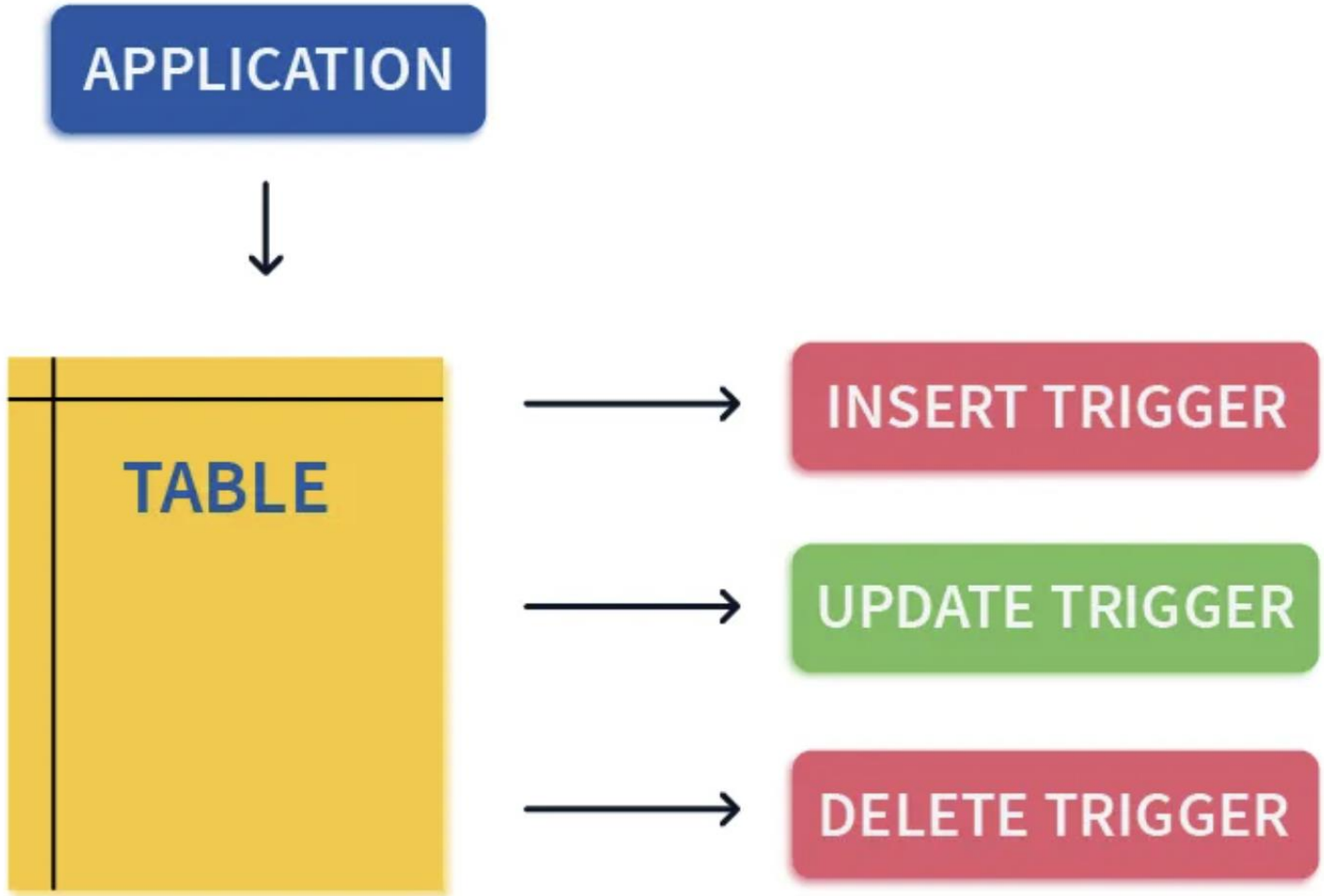
  - *Example*- SQL drops the "Brazil Customers" view:
    - ***DROP VIEW [Brazil Customers];***

Uses of View

- Restricting data access
- Hiding data complexity
- Simplify commands for the user
- Store complex queries
- Rename Columns
- Multiple view facility

Triggers

- A trigger is a database object which fires when an event occurs in a database. For example, a trigger can be set on a record insert in a database table.
- Trigger points of a trigger
  - When any DDL operation is done. E.g., CREATE, ALTER, DROP
  - For a DML operation. e.g., INSERT, UPDATE, DELETE.
  - For a database operation like LOGON, LOGOFF, STARTUP, SHUTDOWN or SERVERERROR


APPLICATION
TABLE
INSERT TRIGGER
UPDATE TRIGGER
DELETE TRIGGER

CREATE Trigger

CREATE [OR REPLACE] TRIGGER trigger_name

{BEFORE | AFTER }

{INSERT [OR] | UPDATE [OR] | DELETE}

ON table_name

[FOR EACH ROW]

WHEN (condition)

[trigger_body]

CREATE Trigger Overview

- CREATE [OR REPLACE] TRIGGER trigger_name: In the first line, we give the name of the trigger we are creating/updating. Here [trigger_name] is to be replaced by the name you want to give to your trigger.
- {BEFORE / AFTER}: Here, we define when the trigger would run, i.e., before or after the DML operation. For example, if we want a trigger to be executed after insertion to a table, we will write after here.
- {INSERT [OR] / UPDATE [OR] / DELETE}: This is the operation or event we have to define, which will trigger a procedure to be executed. If we want a procedure to run after a deletion happens on the table, then we will consider writing delete here.
- on [table_name]: Here, we have to specify the name of the table on which we are attaching the trigger. SQL will listen to changes on this table.
- [for each row]: This line specifies that the procedure will be executed for each one of the rows present. Here we can set a condition for which rows to be impacted. This part is optional, though; in case we don't use this, the trigger shall convert to a "statement-level" trigger rather than being a "row-level" one, i.e., instead of firing the trigger procedure for each row, it will only execute once for each applicable statement.
- WHEN (condition): Here, we mention some condition basis on which the trigger will run. In the absence of a when condition, you can expect it to run for all the eligible rows. This is very important as this will control which rows the trigger must run.
- [trigger_body]: This is the main logic of what to perform once the trigger is fired. In the previous statements, all we defined is when this trigger will be fired, but here we have to define what to do after the trigger is fired. This is the main execution code.

Example

```
CREATE TABLE student(Id integer PRIMARY KEY, first_name varchar(50), last_name varchar(50), full_name varchar(50));


create trigger student_name
after INSERT
on
student
for each row
BEGIN
  UPDATE student set full_name = first_name || ' ' || last_name;
END;
```

Example

```
/* Create a few records in this table */
INSERT INTO student(id, first_name, last_name) VALUES(1,'Alvaro', 'Morte');
INSERT INTO student(id, first_name, last_name) VALUES(2,'Ursula',
'Corbero');
INSERT INTO student(id, first_name, last_name) VALUES(3,'Itziar', 'Ituno');
INSERT INTO student(id, first_name, last_name) VALUES(4,'Pedro', 'Alonso');
INSERT INTO student(id, first_name, last_name) VALUES(5,'Alba', 'Flores');
```

Display Triggers in SQL

- SHOW TRIGGERS LIKE 'stu%’;
- SHOW TRIGGERS **FROM** database_name **WHERE** search_condition
- SHOW TRIGGERS **FROM** database_name;

Drop Triggers in SQL

DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name;

Advantages of Triggers in SQL

- Automation: Triggers automate certain tasks and actions in the database, making it easier to maintain consistency and enforce business rules.
- Improved Data Integrity: Triggers help enforce data integrity by checking for errors or constraints before data is written to the database, helping to prevent data corruption.
- Improved Data Consistency: Triggers can ensure that data is consistent across multiple tables and enforce referential integrity by automatically updating related data.
- Auditing: Triggers can be used to log changes to the data, providing a record of who made changes, when they were made, and what was changed.
- Reusable Code: Triggers allow for reuse of common logic and can reduce the amount of code required to implement complex rules and processes.
- Performance: Triggers can be used to optimize performance by offloading complex processing to the database, reducing the amount of code required in the application.
- Cleaning events – Post someone executes delete statements, we can run certain cleaning logic

Disadvantages of triggers

- Complexity: Triggers can add complexity to a database, making it more difficult to understand and maintain, especially in large and complex systems.
- Performance: Triggers can slow down database performance, especially if they are poorly designed or if they fire too frequently.
- Debugging: Debugging and troubleshooting can be more difficult with triggers, since they are not always easily visible in the code.
- Testing: Triggers can make testing more complex, since they can affect data in unexpected ways.
- Dependencies: Triggers can create dependencies between different parts of the database, making it more difficult to make changes or perform maintenance.
- Security: Triggers can pose a security risk, especially if they are used to execute arbitrary code or to modify sensitive data.
- Portability: Triggers can be difficult to port to different database management systems, since they are not always supported by all systems or may be implemented differently.

Limitations of trigger

- MySQL triggers do not allow to use of all validations; they only provide extended validations. **For example**, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
- Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
- Triggers may increase the overhead of the database server.

- The trigger body can access the column's values, which are affected by the DML statement.
- The **NEW** and **OLD** modifiers are used to distinguish the column values **BEFORE** and **AFTER** the execution of the DML statement.
- We can use the column name with NEW and OLD modifiers as **OLD.col_name** and **NEW.col_name**.

Trigger Event	OLD	NEW
INSERT	No	Yes
UPDATE	Yes	Yes
ELETE	Yes	No

Trigger Examples

OLD AND NEW

```
CREATE TABLE employee(
    name varchar(45) NOT NULL,
    occupation varchar(35) NOT NULL,
    working_date date,
    working_hours varchar(10)
);
```

OLD AND NEW

```
INSERT INTO employee VALUES
('Robin', 'Scientist', '2020-10-04', 12),
('Warner', 'Engineer', '2020-10-04', 10),
('Peter', 'Actor', '2020-10-04', 13),
('Marco', 'Doctor', '2020-10-04', 14),
('Brayden', 'Teacher', '2020-10-04', 12),
('Antonio', 'Business', '2020-10-04', 11);
```

OLD AND NEW

```
Create Trigger before_insert_empworkinghours
BEFORE INSERT ON employee FOR EACH ROW
BEGIN
IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;
END IF;
END //

INSERT INTO employee VALUES
('Alexander', 'Actor', '2020-10-012', -13)
```

MySQL: AFTER DELETE Trigger

- An AFTER DELETE Trigger means that MySQL will fire this trigger after the DELETE operation is executed.

```
CREATE TRIGGER trigger_name
AFTER DELETE
  ON table_name FOR EACH ROW
BEGIN
  -- variable declarations
  -- trigger code
END;
```

MySQL: AFTER DELETE Trigger

```
CREATE TABLE contacts
( contact_id INT(11) NOT NULL AUTO_INCREMENT,
  last_name VARCHAR(30) NOT NULL,
  first_name VARCHAR(25),
  birthday DATE,
  created_date DATE,
  created_by VARCHAR(30),
  CONSTRAINT contacts_pk PRIMARY KEY (contact_id)
);
```

```
DELIMITER //
CREATE TRIGGER contacts_after_delete
AFTER DELETE
  ON contacts FOR EACH ROW
BEGIN
  DECLARE vUser varchar(50);
  -- Find username of person performing the DELETE into table
  SELECT USER() INTO vUser;
  -- Insert record into audit table
  INSERT INTO contacts_audit
  ( contact_id,
    deleted_date,
    deleted_by)
  VALUES
  ( OLD.contact_id,
    SYSDATE(),
    vUser );
END; //
DELIMITER ;
```

MySQL: AFTER INSERT Trigger

An AFTER INSERT Trigger means that MySQL will fire this trigger after the INSERT operation is executed.

```
CREATE TRIGGER trigger_name
AFTER INSERT
   ON table_name FOR EACH ROW
BEGIN
   -- variable declarations
   -- trigger code
END;
```

```
DELIMITER //
CREATE TRIGGER contacts_after_insert
AFTER INSERT
  ON contacts FOR EACH ROW
BEGIN
  DECLARE vUser varchar(50);
  -- Find username of person performing the INSERT into table
  SELECT USER() INTO vUser;
  -- Insert record into audit table
  INSERT INTO contacts_audit
  ( contact_id,
    created_date,
    created_by)
  VALUES
  ( NEW.contact_id,
    SYSDATE(),
    vUser );
END; //
DELIMITER ;
```

Connect with me on LinkedIn

Please subscribe to our YouTube channel

Check out my GitHub profile

Follow me on Twitter(X)

Thank you